Josiah Schmitz

CS 3100

Project 3.1 - MultiSet Design

**1. Introduction**

This MultiSet ADT will be made using my custom hash table implementation from Project 1. This specific hash table uses simple hashing with linked lists that contain buckets holding key-value pairs. The implementation will be slightly altered with override methods to work with just **std::string** elements instead. The MultiSet will support basic operations such as insertion, removal, contains/count checks, and console print output, as well as set operations such as union, intersection, difference, and symmetric difference. An extension feature, Serialization/Deserialization, will also be added to allow for saving and loading the multiset to/from a file.

**2. Data Structure Choice**

**Chosen Structure:** HashTable with Linked List Chaining (see Project 1 submission)

**Justification:**

* The existing hash table class (HashTable) has efficient insert, remove, and lookup methods because of its use of hashing and chaining. Subsequently, this will allow the MultiSet to run very efficiently.
* Managing duplicates is simply done by allowing each unique key to map to a count that represents how many times it appears.
* Set operations, like union and intersection, can be implemented very easily by just comparing or applying basic math operations on the keys’ counts.
* The chaining method makes potential resizing of the MultiSet very simple and efficient.
* Compared to an AVL tree (see Project 2 submission), the hash table doesn’t need the extra data management that comes with balancing, rotations, and traversal logic.

**3. Core MultiSet Operations**

**insert(const std::string& key, size\_t num = 1):**

* Find key in the hash table using **contains()** method.
* If key exists, increment the count by **num**.
* If it doesn’t exist, insert new entry with count **num**.

**remove(const std::string& key, size\_t num = 1):**

* Find key in the hash table using **contains()** method.
* If **count** >= **num**, subtract **num** from **count**. If result is zero, remove the key entirely.
* If **count** < **num**, do not modify hash table and return **false**.

**remove(size\_t num = 1):**

* Remove first **num** entries from the table.
* Collect the removed elements in a vector.
* Removal will start at the front of each non-empty bucket and remove first **num** elements.
* If **num** >= **count** of first key, completely remove key and continue removing next entry.

**contains(const std::string& key):**

* Loops through all buckets in hash table and checks if **key** equals each bucket’s key.
* Return **true** if key is found in the table.

**count(const std::string& key):**

* Return the current count for the given key.

**keys():**

* Loop through the hash table and add each key **count** times to a vector.
* Return a vector with each key repeated **count** times.

**uniqueKeys():**

* Return a vector with all keys in the table once each, ignoring **count** this time.

**empty():**

* Return true if the multiset has no elements (meaning the hash table’s size is zero).

**size():**

* Loop through the hash table and add each key’s count to **size** variable.
* Return size, which contains total number of all elements.

**uniqueSize():**

* Loop through hash table and add one to **uniqueSize** variable for each key encountered.
* Return **uniqueSize**.

**clear():**

* Reset the internal hash table and remove all data by completely reinitializing the table.

**operator<<:**

* Loop through the hash table and print out each entry in form {key: count}.
* (May possibly create/use helper method to put all text into one string for **operator<<**).
* Note: created helper method printMe(); see in program.

**4. Set Operations**

**unionWith(const MultiSet& other):**

* For each key in both sets, take the maximum **count**.
* Merge results into a new MultiSet.

**intersectionWith(const MultiSet& other):**

* For any shared keys in both sets, use the minimum **count** from both sets.

**differenceWith(const MultiSet& other):**

* For each key in the base set, subtract the **count** in other set.
* Only include the key if the result is greater than 0.

**symmetricDifferenceWith(const MultiSet& other):**

* Combine (A - B) and (B - A) into one result.

**5. Anticipated Challenges**

* Making sure I can efficiently manage and update duplicate counts.
* Being able to resize the MultiSet without losing key-count mappings.
* Implementing the remove(size\_t num) method in a consistent and repeatable way.
* Preventing data corruption/modification during deserialization.
* Writing clean logic for the set operations.
* Creating an efficient print method to display the MultiSet to the console.

**6. Extension Feature Plan: Serialization/Deserialization**

This will allow for saving a MultiSet to a text file and recreating it at a future point.

(All method names subject to change. This is just a general idea of the methods I’ll need, and I may need to add additional methods if I encounter additional problems.)

**saveToFile(const std::string& filename):**

* Create an output stream and iterate over all unique keys.
* Write each key and its count on a new line in the format: {**key** **count**}
* Will work somewhat similarly to my print helper method

**loadFromFile(const std::string& filename):**

* Wipe the current MultiSet.
* Loop over reading each line and split it into **key** and **count**.
* Insert the **key** with the given **count** into the MultiSet.